On the Formal Verification of Middleware Behavioral Properties

Jérôme Hugues, Thomas Vergnaud, Laurent Pautet [1]

GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France

Yann Thierry-Mieg, Soheib Baarir, Fabrice Kordon [2]

Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/SRC
4, place Jussieu, F-75252 Paris CEDEX 05, France

Abstract

Distribution middleware is often integrated as a COTS, providing distribution facilities for critical, embedded or large-scale applications. So far, typical middleware does not come with a complete analysis of their behavioral properties. In this paper, we present our work on middleware modeling and the verification of its behavioral properties; the study is applied to our middleware architecture: PolyORB. Then we present the tools and techniques deployed to actually verify the behavioral properties of our model: Petri nets, temporal logic and advanced algorithms to reduce the size of the state space. Finally, we detail some properties we verify and assess our methodology.

Key words: Middleware design, Verification, Petri nets, LTL, Symmetries

1 Introduction: Issues in Middleware Engineering

Distribution middleware is now widely integrated as *Components Off-The-Shelf* (COTS) in Distributed Real-Time Embedded (DRE) systems. The properties of each building block must be known to ensure its correct integration to such system [4] and to ensure the correctness of the system as a whole.

In this context, the European Spatial Agency (ESA) identified several use cases for middleware. They include ground stations interacting with satellites as well as fleets of collaborating satellites and drones. These applications require multiple

[1] Email:{jerome.hugues, thomas.vergnaud, laurent.pautet}@enst.fr

[2] Email:{yann.thierry-mieg, souheib.baarir, fabrice.kordon}@lip6.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

distribution mechanisms to handle variations in communication channels, flexible resource management, and ensure autonomy for long missions.

In addition to distribution needs, these systems come with non-functional requirements, inherited from real-time engineering such as reliability, availability, dependability. Hence, properties (like determinism, safety, liveness, timeliness) must be verified during the design process and in particular at the middleware level.

Middleware solutions now support the requirements of most distributed applications. They usually supports one given distribution model: a combination of one or several mechanisms to enable distribution, e.g. Message Passing (MP), Remote Procedure Call (RPC), Distributed Object Computing (DOC) or Shared Memory (SM). Yet, they usually do not address the verification of key behavioral properties such as request fairness, absence of deadlock or correct resource dimensioning.

This calls for a next-generation of middleware that addresses these many challenges [20]. Middleware architecture should be versatile to meet application needs. Moreover it should follow an extensive proof-based system engineering approach to provide strong evidence it is correct with respect to application requirements.

This paper details a joint work on middleware verification lead by the CS department of the ENST (middleware experts) and SRC/LIP6 (verification experts).

We first review well-known existing middleware architectures, and show they do not facilitate verification; then, we introduce the schizophrenic middleware architecture as a solution to achieve verification. To verify key behavioral properties in our middleware, we refine our middleware components, we model them using Petri nets and we assemble models to build one middleware configuration. As the system may be very complex, we present the algorithms we used to address state space explosion and apply them to check middleware behavioral properties. This provides a first step towards building proof-based middleware.

2 Problem statement

In this paper, we focus on the complete characterization of distribution middleware. In this section, we present some background on middleware architectures. We then discuss the use of formal methods to verify behavioral properties of middleware.

Formal modeling and middleware engineering are usually considered as two different expert domains. For instance, they are considered either by separate teams of the project or at separate steps in the design process. To efficiently reach our objective, we propose to reconcile system modeling and middleware engineering, and join ENST and LIP6 efforts. The ENST has a long experience in middleware implementations, including GLADE, the only industrial implementations of the Distributed System Annex of Ada 95; AdaBroker an Open Source CORBA ORB. The LIP6/SRC department has a long experience in the development of algorithms and tools to apply formal methods to distributed systems, it has developed the CPN-AMI tool-suite to model and then analyse systems using Petri nets.

2.1 *Middleware for DRE systems*

DRE systems require adaptable distribution models and middleware architectures: Rajkumar [19] advertised Message Passing as a solution for DRE systems, and proposed the Real-Time Publisher/Subscriber service. The Ada Distributed Systems Annex [13,17] integrates several mechanisms (DOC, RPC, SM). RT-CORBA extends CORBA's DOC mechanisms for real-time systems and integrates support for many QoS policies [21]. This leads to several tailorable middleware architectures.

For instance, *configurable middleware*, such as TAO [22], let applications select specific run-time policies to support the DOC distribution model. It relies on architectural and design patterns [9] to support a large number of policies.

Adaptive and reflective middleware [20,3] extends middleware configurability mechanisms to enable adaptability to specific changes in application context. This architecture provides promising properties to meet QoS applications requirements.

Generic middleware, such as Jonathan [26], defines abstract canonical components and architecture, their instantiation provides a specific distribution model. Jonathan provides a CORBA personality (*David*), a Java RMI personality (*Jeremie*) and specialized personalities for multimedia systems.

These architectures can meet stringent requirements. But they only partially determine their properties: they usually rely on the testing of specific scenarios such as Boeing's Bold Stroke OFP [23]. However, there is a double combinatorial explosion when considering middleware as a whole: the number of possible execution scenarios for one middleware configuration increases with the interleaving of threads and requests; the number of possible configurations increases with middleware adaptability and versatility.

Thus, we claim testing is not sufficient to assess middleware behavioral properties such as *absence of deadlocks*, *request fairness*, or *correct resource dimensioning*. We propose to use formal methods to model and then verify our system.

2.2 *Formal methods for middleware verification*

This sections discusses the choice of a modeling language and its formal verification methods that are appropriate to model and then verify middleware.

There are currently two families of formal methods: proof-based verification (such as B [1] or Z [8]) and model checking (using tools and languages such as SPIN [11] or LUSTRE [10]). In proof-based methods, the model is described by means of axioms, properties are theorems to be verified using a theorem prover. In model checking, the model is expressed using a language from which an exhaustive execution can be computed (this usually requires a mathematically based definition). An "execution engine" produces the exhaustive state space associated to the system as a graph where actions (atomic instructions in the language) relate to states (a given possible value of the system's context). It is then possible to explore the graph to check if a property is satisfied.

These two approaches are complementary. Proof-based techniques allow the analysis of infinite systems. However, the use of a theorem prover is a very difficult and a very technical task that is hard to automate. On the contrary, model checking is dedicated to finite-state systems but modeling and verification can be done using graphical toolkits and most steps can be automated [6].

We selected *Well-formed Petri nets (WN)* [5] as an input language for model checking. WN are high-level Petri nets, in which tokens are typed data holders. This allows for a concise and parametric definition of a system, while preserving its semantics. One main feature of *WN* is that they allow the automatic and efficient construction of the *symbolic reachability graph*: a quotient state-space, in which nodes are equivalence classes of states, and arcs equivalence classes of events. This graph is built by exploiting symmetry, yielding a symbolic state-space sometimes exponentially smaller than the concrete state-space, and thus more manageable.

Recent works have automated the analysis of the symmetries allowed by a system [25], and of the symmetries allowed by a property [2], allowing full LTL model-checking while efficiently fighting the well-known combinatorial state-space explosion problem. As we show in sections 5.2 and 6.1, the use of these symmetry-based reductions allows us to verify the properties of our system, while plain state-space generation is unfeasible with classic techniques.

In the remainder of the paper, we present the Schizophrenic Middleware architecture, and show how we reengineer it to enable the verification of middleware. Then, we detail the algorithms we used and analyze our middleware architecture.

3 The Schizophrenic Middleware Architecture

In this section, we introduce the key elements of the "schizophrenic middleware architecture"; and present its role in the verification of behavioral properties.

Middleware combines two complementary facets: (1) a framework to implement distributed systems, using the host and operating system resources (e.g. tasks, I/O); and (2) a set of services to build portable distributed applications. In [12], we introduced the "schizophrenic middleware" architecture: a unique architecture that advertises these two aspects, and enforces separation of concerns.

In [28], we present PolyORB, our implementation of such a schizophrenic middleware. We assess its suitability as a middleware platform to support multiple specifications (CORBA, Ada Distributed Systems Annex, Web Applications, Ada Messaging Service close to Sun's JMS) and as a COTS for industry projects [3].

From our experiments, we note that a reduced set of services is sufficient to describe various distribution models. We identify seven steps in request processing, each of which is defined as one fundamental service. Services are generic components for which an general implementation is provided. Developers may provide an alternate implementation. Each middleware instance is one coherent assembling of these entities. The μBroker component coordinates these different services, it is responsible for the correct request propagation in the middleware instance.

[3] PolyORB is supported by AdaCore (`http://libre.act-europe.fr/polyorb`)

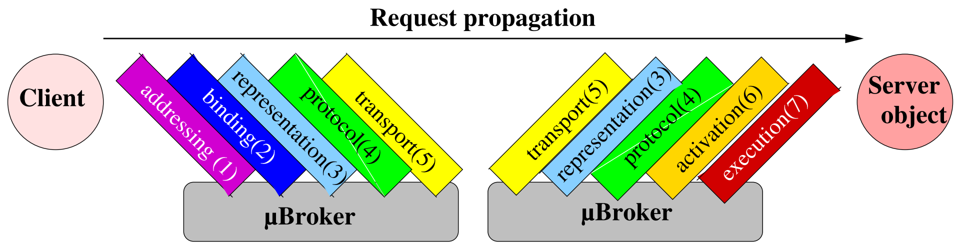

Fig. 1. Request propagation in the Schizophrenic Middleware architecture

Figure 1 illustrates how PolyORB services cooperate to transmit one request between two application entities, located on two separate nodes.

The client looks up server's reference using the *addressing* service (1), a dictionary-like component. Then, it uses the *binding* factory (2) to establish a connection with the server, using one communication channels (e.g. sockets, protocol stack).

Request parameters are mapped onto a representation suitable for transmission over network, using the *representation* service (3), this is a mathematical mapping that convert a data into a byte stream (e.g. CORBA CDR).

A *protocol* (4) is implemented for transmissions between the client and the server nodes, through the *transport* (5) service, which establishes a communication channel between the two nodes. Both can be reduced to *automata*. Then the request is sent through the network and unmarshalled by the server.

Upon the reception of a request, the middleware instance ensures that a concrete entity is available to execute the request, using the *activation* service (6). Finally, the *execution* service (7) assigns execution resources to process the request. These two services rely on the *factory* and *resource management* design patterns.

Hence, services in our middleware architecture are *pipes and filters*: they compute a value and pass it to another component. Our experiments with PolyORB showed they follow the same semantics, they are only adapted to match precise specifications. They can be reduced to well-known abstractions.

The μBroker handles the coordination of these services: it allocates resources and ensures the propagation of data through middleware. Besides, it is the only component that controls the whole middleware: it manipulates critical resources such as tasks and I/Os or global locks. It holds middleware behavioral properties.

The Schizophrenic Middleware architecture provides a comprehensive description of middleware. This architecture separates a set of generic services dedicated to request processing from the μBroker. The latter is directly responsible for middleware behavior. Thus, we isolate the control loop of our system, present in all middleware instances: it is the key component to be verified first.

4 μBroker: source code and formal model implementations

We identified the μBroker as the control loop of our architecture. We now discuss its architecture and mapping to both source code and formal implementations.

4.1 *µBroker architecture*

We first propose an architecture for the µBroker, and detail its components.

Several "strategies" have been defined to create and use middleware resources: [18] detail different request processing policies implemented in TAO; the CARISM project [14] allows for the dynamic reconfiguration of communication stacks. Hence, the µBroker must be adaptable enough to support most of them, and still provides a clear design to enable modeling and then verification.

We propose the following architecture for the µBroker(figure 2):

- the *µBroker Core API* handles the functional parts of the interactions with other middleware services; it provides an interface to configure the middleware instance and helper routines to execute specific functions such as managing I/O. This component interacts directly with middleware the *Binding*, *Transport*, *Execution* and *Addressing* services;
- the *µBroker Controller* manages the state automaton associated to the µBroker. It grants access to middleware internals (tasks, I/O and queues) and schedules tasks to process requests. It is responsible for the behavioral part of the µBroker. Several policies refines its behavior: the *Asynchronous Event Checking* policy sets up the polling and read strategies to check events on I/O sources; the *Request Scheduler* sorts request to be processed (e.g. FIFO, EDF orders), the *Dispatcher* selects threads that execute requests.

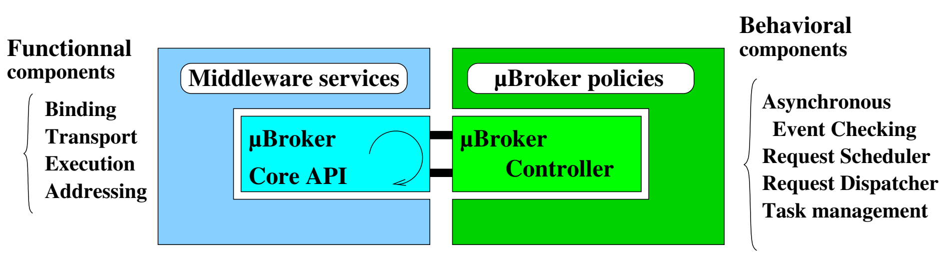

Fig. 2. The two sides of the µBroker

µBroker entities are defined by their interface and a common high-level behavioral contract, instances of these entities may refine this behavior to support different policies. This architecture has been implemented in PolyORB. It uses well-defined entities, and demonstrates its adaptability to support classical policies found in configurable middleware or defined in specifications such as RT-CORBA.

The µBroker pattern proposes a comprehensive description of the middleware control loop, and a step towards verification of middleware behavioral properties.

4.2 *Modeling one middleware configuration*

We now describe the modeling of our architecture using Petri nets as a language for system modeling and verification (figure 3).

Step 1: we elaborate one Petri net module for each middleware components variation. Petri net transitions represent atomic actions; Petri net places are either

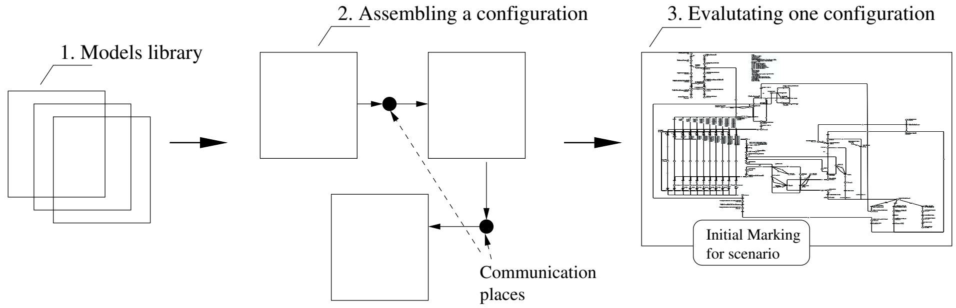

Fig. 3. Steps of the μBroker modeling

middleware states or resources. Common places between different modules define interactions between Petri nets modules, they act as *channel places* [24].

Step 2: for one configuration of the μBroker, some Petri net modules are selected to produce the complete model. Communications places (outlined in black) represent links to other μBroker functions or to middleware services.

Step 3: the selected modules are merged to produce a global model. This model and one initial marking enable the verification of the middleware properties.

Functions can be separately verified and then combined to form the complete Petri net model. Multiple models can be assembled from a common library of models. Thus, we can test for specific conditions (policies and settings).

The initial marking of the Petri Net defines available resources (e.g. threads, I/O); or sets up internal counters. Its state space covers all possible interleaving of atomic actions: we test all possible execution orders.

We have detailed the steps from middleware requirements for DRE systems, down to the modeling of one configuration using Petri nets. This allows us to verify specific middleware behavioral properties on our models.

5 Verifying properties of the μBroker

In this section, we introduce some of the expected properties the μBroker as well as the formal techniques used to verify them by model-checking.

5.1 μBroker configurations and models

In this section, we review the key parameters that characterize the μBroker, and some of the properties one might expect from such a component.

The μBroker is defined by the set of policies and the resources it uses. These settings are common to a large set of applications. We consider one middleware instance, in server mode, that processes all incoming requests. We study two configurations of the μBroker: *Mono-Tasking* (one main environment task) and *Multi-Tasking* (multiple tasks, using the Leader/Followers policy described in [18]). The latter allows parallel request processing.

We assume that middleware resources are pre-allocated: we consider a static pool of threads; a bounded number of I/O sources and one preallocated memory pool to store requests. This hypothesis is acceptable: it corresponds to typical engineering practices in the context of critical systems. Our implementations and the corresponding models are controlled by three parameters:

- S_{max} is the upper bound of I/O Sources listening for incoming data;
- T_{max} is the number of Threads available within the middleware;
- B_{size} is the size of the Buffer allocated to read data from I/O sources.

S_{max} and T_{max} define a workload profile for the middleware node, B_{size} defines constraints on the memory allocated by the μBroker to process requests. These parameters control middleware throughput and execution correctness.

We list four essential properties of our component. They represent key properties our component must verify to fulfill its role.

- $P0$ (symmetry): threads and I/O are unordered, elements in a class are equivalent;
- $P1$ (no deadlock): the system may always process incoming requests;
- $P2$ (consistency): there is no buffer overflow;
- $P3$ (fairness): every event on a source is detected and processed.

$P0$, $P1$, $P3$ are difficult to verify only through the execution of some test cases: one has to examine all possible execution orders. This may not be affordable or even possible due to the many possible threads and requests interleaving. Besides, the adequate dimensioning of static resources to ensure consistency ($P2$) is a strong requirement for DRE systems, yet it is a hard problem for open systems such as middleware. Thus, we propose to verify them by model-checking.

5.2 Analysis methods

The system we model is complex and uses different resources, its state-space is expected to be huge. We detail the figures in section 6.1: the system has a state-space of up to 10^{11} states for the values we considered and is thus impossible to treat with classic methods. We therefore decided to reduce the problem by performing a symmetry analysis of the model, and using the results to generate a compact representation of the state-space of the system, suitable for model-checking. This symmetry analysis is fully automated, and is performed for each given configuration of the middleware we considered. We also take into account the property to be checked, allowing further reductions by abstracting away behaviors that are not observed by the property, using dynamically adapted on-the-fly algorithms. This section superficially explains how the tools work (complete descriptions of the algorithms can be found in [25,2]). Moreover, these techniques may be used through a simple interface, that requires no knowledge of these internals.

5.2.1 P0: Analyzing the symmetries of the model

We wish to prove property $P0$ (symmetry) ; the first step is a symmetry analysis that explores the model's structure to determine which data types present a homogeneous behavior. Two elements e_1 and e_2 of a given data type are symmetric if exchanging e_1 with e_2 at any point in the execution of the system does not modify its behavior. By behavior, we mean that *from an observer's point of view*, the system appears to be the same.

For this analysis, we first consider the LTL point of view of an *action-based* observer that sees all events happening as a sequence. To this slightly myopic observer, the actual values used to trigger the events are blurred, only the sequences of occurrences of actions are visible. Such a sequence is called an ω-word, the alphabet of which is an occurrence of an action. The set of ω-words that the system can generate using the alphabet of event occurrences is called the *language* of the *actions* of the system. Due to interlacement semantics, this language may be exponentially smaller than the language of the system over an alphabet that sees operation arguments. It can be represented by a *quotient* state-space graph, thus exponentially smaller than the concrete state-space graph.

To analyze the symmetries of a system, our algorithm [25] examines each action (transition) of the system to find which elements (if any) it distinguishes, and concludes that any element distinguished by at least one action should be distinguished in $\mathscr{R}$. The result of this symmetry analysis is an *equivalence relation* $\mathscr{R}$ that lists for each data type the symmetric values with respect to the language of the actions of the system. This automatic computation ensures that any elements considered equivalent under $\mathscr{R}$ are not distinguished by *any* action of the system. $\mathscr{R}$ defines a set allowable permutations or rotations [5] that can be applied to the objects of the system without affecting the transition relation.

We then translate high-level annotations to symbolically expressed ones that use $\mathscr{R}$ in a simple and readily analyzable form, the rigorous syntax of Well-Formed nets [5]. Finally we construct a *quotient reachability graph under* $\mathscr{R}$, using the GreatSPN [7] kernel, in which nodes represent an equivalence class of states under $\mathscr{R}$. Let us mention that computation of symmetries is very fast since it is a *structural property* of the Petri net. Its complexity is thus related to the size of the specification (number of places and transitions), but not to the state space size.

Symmetry analysis of the μBroker is computed on the structure of our models. It yields an equivalence relation $\mathscr{R}$ that states that all Threads (resp. Sources) are equivalent. Therefore, we prove that $P0$ *(symmetry)* is true. Note that this property was not true on the first versions of the model, and that to obtain it we had to adapt some modeling choices of the components of the μBroker.

5.2.2 P1, P2: Verifying symmetric properties

The property $P1$ states that there exists no deadlock; this is a property that our action-based LTL observer can see. We therefore can verify this property on the quotient produced under the relation $\mathscr{R}$. The quotient graph lists at most $S_{max}! \cdot T_{max}!$ permutations for each state in a compact manner, it is thus exponentially smaller than the concrete state-space, allowing faster verification of the property.

To verify $P2$ *(consistency)*, we check that accesses to memory pools are correct. The places $(\mathcal{DataSlots}_i)_{i \in 1..M}$ represent the memory pool. Write operations insert a token in one place from this set, read operations withdraw one. Data inconsistency occurs when writing more than once in a slot. It is tested by a safety property expressed by an LTL formula (1), that asserts that such a state is unreachable.

$$\forall d \in \mathcal{DataSlots}, \mathsf{G}(card(d) \leq 1) \tag{1}$$

The interesting point is that this property is directly observable on the graph produced under $\mathscr{R}$, as permutations of elements will not change the cardinality of the place that is tested in $P1$. Therefore if we consider a node of the quotient state-space, as all its elements are equivalent under $\mathscr{R}$, we can assert whether or not it verifies $card(d) \leq 1$ directly. An analysis of the symmetries of the property proves and uses this property automatically, without user's intervention.

Thus, we verify there is no data corruption for $S_{max} \leq B_{size}$, by model checking for different values of $S_{max}$and B_{size}, but we notice that data-corruption occurs otherwise. We interpret this result by the fact that model semantics does not allow more than $card(\mathcal{DataSlots})$ successive write operations, and thus does not overflow the buffer.

5.2.3 *P3: Exploiting* observed *behavioral symmetries*

We have seen the equivalence relation of the system $\mathscr{R}$, although permissive, can allow verification of symmetric properties. However, to verify $P3$ *(fairness)*, we need to evaluate the LTL formula:

$$\begin{cases} \forall s \in \mathcal{Sources}, a = \{s\} \subseteq \texttt{ModifiedSrc}, b = \{s\} \subseteq \texttt{DataOnSrc}, \\ \qquad\qquad \mathsf{G}(a \Rightarrow \mathsf{F}\, b) \end{cases} \tag{2}$$

It states that any event on a given source s will eventually be handled. If so, the token representing s should move from the notification place `ModifiedSrc` ("the μBroker detected a pending request") to initial place `DataOnSrc` ("new incoming request"). This property cannot be observed reliably by our action-based LTL observer, as it requires that the binding of s in the action "the μBroker detected a pending request from s" be remembered, so that we can verify that the same s is eventually seen moving to `DataOnSrc`. However, thanks to symmetry, if $\exists s \in \mathcal{Sources}$ such that $P3$ holds true, as we know all sources to be equivalent, we will have proved the property $\forall s \in \mathcal{Sources}$. This avoids redundant verification for each source, although this would also be possible as the number of sources is supposed finite.

We therefore choose a source s that we individualize, by constructing a quotient graph under more restrictive relation $\mathscr{R}'$ such that s cannot be permuted with other sources. Thus our observer can now see events relating to s as different from events relating to other sources. However, the cost of distinguishing even a single source is up to a factorial increase ($card(\mathcal{Sources})!$) in the state-space size. Thus even if the system is structurally symmetric, when the formula is not –as in this case– its evaluation generates again (though with a delay) the state-space explosion problem.

Thus, the quotient reachability graph suffers from a major limitation: it does not tackle *partially symmetric systems* or *partially symmetric properties*. However recent advances have shown that these partial symmetries can be exploited in LTL verification, through the Symbolic Synchronized Product (SSP) method [2]. The basic intuition is to use an observer capable of *zooming in* (i.e. by using a refined $\mathscr{R}'$) whenever there is a potential problem, and *zooming out* (i.e. by allowing more permutations in $\mathscr{R}'$) when a "blurred" vision is enough to verify the property.

For instance, the property may distinguish objects only in *parts of an execution run*, e.g. $P3$. Sources can be permuted when there are no requests pending. We need to distinguish the source s only when the place `ModifiedSrc` contains a request from a source s. This partial temporal asymmetry can be captured by working with the Büchi automaton that represents the LTL property [27]. To perform model-checking, we compute the intersection of the language recognized by the *negation* of a property with the language of the system. This operation is computed by the *synchronized product* of the state-space and the automata of the formula. It is thus usually of complexity polynomial over the product of the sizes of the automaton of the LTL formula, and of the state-space itself. However, we construct our symmetry-aware Symbolic Synchronized Product on the fly, and we adapt the relation $\mathscr{R}'$ (our zoom setting) under which we build the successors of a state, to the arc of the LTL automaton with which it is being synchronized.

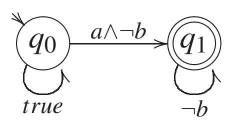

Fig. 4: Automata of the negation of property $P3$

Consider the automata corresponding to the negation of $P3$ represented figure 4. An ω-word (or sequence of events) recognized by this automaton is composed of a finite sequence of events (we don't care which, synchronizing with the arc true), followed by an event such that $a \wedge \neg b$ becomes true after the event, and then $\neg b$ always remain true (accepting state q_1). Such a sequence contradicts $P3$, thus producing a counter-example. In the SSP construction, while synchronizing with the arc "true", no asymmetry is introduced thus all model objects are equivalent, we can zoom out; but a synchronization with the arc " $a \wedge \neg b$" forces to individualize the source "s" (zoom in) to allow a correct symbolic verification.

In the current case, as $P3$ holds true, the sequences of the synchronized product that match "$a \wedge \neg b$" but not "$\neg b$ always true" are eventually garbage collected. The final size of the SSP proving the validity of $P3$ is therefore no more than the size of the quotient graph under $\mathscr{R}$ allowing all permutations, as all actions in the SSP end up synchronizing with the arc $true$.

The SSP is a powerful technique, also able to tackle partially symmetric systems, in which objects behave asymmetrically only in localized parts of the model. Usually, as the action-based symmetry analysis is global, these local asymmetries will yield a restrictive relation $\mathscr{R}$, and the graph built under $\mathscr{R}$ may be as large as the concrete state-space. A typically encountered case is distributed systems that present an initialization phase in which privileged initiators play a different role, but then all participants behave like symmetrical peers once the system is running.

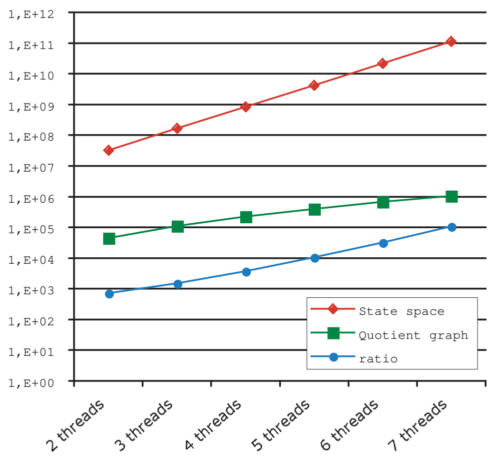

Fig. 5. μBroker's state space and quotient reachability graph for $S_{max}=4$ and $B_{size}=5$.

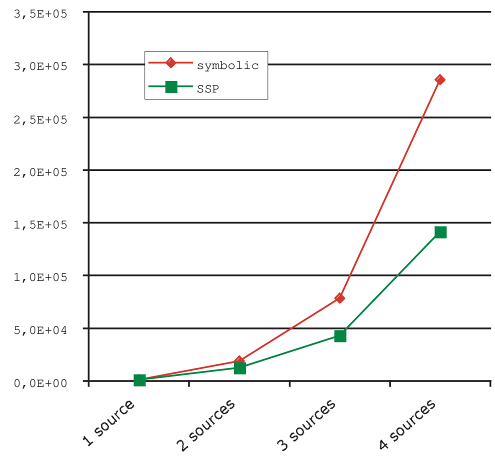

Fig. 6. Number of nodes examined to evaluate $P3$ when $T_{max}=2$ and $B_{size}=4$.

SSP addresses this problem (by zooming in just when running the initialization), but as the μBroker is (by design) fully symmetric, this particular reduction was not used for our verification. We plan to use this capability in further works, for more advanced and complex configurations.

6 Verification results and performances

In this section, we detail figures and performance of the verification of the μBroker.

6.1 Modeling and analysing the μBroker

The *Mono-Tasking* model has 47 places, 38 transitions and 134 arcs. *Multi-Tasking* model is twice as large. Number of tasks and I/Os has no effect on the size of the model (it is coded as types for Petri net tokens), the size of the buffer has an impact on the number of places and transitions.

Figure 5 summarizes the size of the state space for $B_{size}=5$ and $S_{max}=4$. Petri Net models have an average size, yet the state space is large, it denotes the complexity of the system. Its size increases exponentially with T_{max} and S_{max}. We also note that the symbolic reachability graph is smaller by several powers of ten, and that this ration increases with T_{max}. Thus, computations can be completed within acceptable delays on usual workstations, in less than 10 hours for the biggest models, on 2.6GHz Pentium-4 computers with 512MB of memory running GNU/Linux.

From these different figures, we claim that the analysis of PolyORB could not have been performed without the use of these advanced model checking techniques because of intrinsic memory limitations of typical workstations. This is depicted in figure 5. It clearly shows an exponential ratio between the concrete state space and the quotient reachability graph. As an illustration, even for common middleware configurations (7 threads and 4 I/O sources) the system presents about 10^{11} states, but we could compute and evaluate its properties on a quotient reachability graph on a computer having 512MB of memory without swapping. We also observe that the quotient graph evolution decreases. As an example, from 4 to 5 threads, the quotient graph increases by a factor of 80% and from 5 to 6, by a factor of 57%. At a given stage, it should reach an asymptote (e.g. when adding new threads does not impact on the system symbolic complexity at all). This is of particular interest since the state space evolution remains exponential.

The SSP technique brings further reductions as illustrated by figure 6, that compares it to the global symmetry approach. It shows how the gain from the reduction increases as the parameters increase. It is equivalent to the global symmetry approach for one source (no symmetry to exploit), then offers a polynomial size reduction over S_{max}. No value is provided when S_{max} reaches 5: B_{size} is 4 and the property is not verified as we have shown in the previous section, thus the the on-the-fly counter-example production algorithm interrupts state-space examination.

«««< .mine ======= We verified two configurations: we tested LTL formulae representing expected properties, we extracted scenarios that help dimensioning resources. Our model-checking tools overcome the complexity of the μBroker model. The verified properties provide strong evidence that our architecture is correct. »»»> .r810

6.2 *Assessment of our verification tools*

The tools we used have a significant role to achieve verification. Our models were created using CPN-AMI [15], a Petri net CASE environment that integrates structural analysis tools and model checkers. We then analyse data type symmetries [25] and compute the reduced state space using the GreatSPN [7] tool-suite and extensions for SSP. Finally for LTL model-checking, we used the model checking library Spot [16], in conjunction with GreatSPN. All these tools have drivers available that make them plug-ins, homogeneously accessible from CPN-AMI, and hiding the complexity of their interactions from the end-user. CPN-AMI and SPOT are developed in LIP6/SRC. GreatSPN is developed at the Dipartimento di Informatica at the University of Torino. The LTL capabilities of GreatSPN and SSP computation are the result of a cooperation between Paris and Torino.

An important aspect of the verification process is that both the global structural analysis of the system and the partial symmetry analysis used in SSP were computed wholly automatically, thanks to our recently introduced extensions to the GreatSPN kernel [25,2]. As we tested our properties over a wide range of configurations and parameter values, symmetry analysis "by hand" would have been unfeasible, and less reliable as it would have required human intervention.

Our symmetry-aware algorithms help us to address the state-space explosion. The reduced state-space is smaller than the ones produced by typical algorithms by a power of ten. This difference increases with the parameters S_{max}, T_{max}, B_{size}. This demonstrates that our tools make it possible to verify complex systems. The use of formal verification techniques on the μBroker allowed to correct some bugs, and gives us a much higher level of confidence in the architecture of PolyORB.

7 Conclusion

In this paper, we focused on middleware architectures and the verification of their behavioral properties. We outlined current trends in middleware engineering: multiple architectures exist to support the requirements of DRE systems; yet, verification is seldom contemplated. Indeed, verifying a distributed infrastructure is a complex task as long as middleware development does not integrate very early in the design process both distribution functions and verification requirements.

We aim at providing a proof-based middleware. We present the Schizophrenic Middleware Architecture, and detailed steps to verify it. We chose Petri nets as a modeling language, and LTL to express and then verify its properties.

The schizophrenic architecture emphasizes on the separation of concerns in middleware: a set of fundamental services covers middleware functional components, a Middleware Main Loop coordinates them. Services can be adapted to support configurability and genericity requirements. thus, our schizophrenic middleware PolyORB is a good candidate to build proof-based middleware.

We identified the key component that controls the whole middleware behavior and refined it to define the μBroker pattern. This adaptable component proposes a complete definition of a middleware main control loop.

We detailed the steps to model and then verify it using Petri nets, a complete theoretical framework to assess properties of control systems such as the μBroker. Then, we discussed the verification of several configurations of the μBroker.

We introduced tools and algorithms to achieve verification. The state-space explosion is a typical problems when modeling system with Petri nets. We present advanced techniques used to overcome this problem. These techniques allows the testing of LTL formulas on reduced state space: the quotient state space. It fully exploits global and local symmetries of the model. Hence, we could assessed our model is fair for incoming requests, without deadlock; we detailed how we can check that resources are correctly dimensioned. We note that the ratio between the quotient graph and the complete state space is exponential; the state space evolution remains exponential since the quotient graph tends to an asymptote. This analysis could have never been performed using classical model checking techniques.

Thus, we achieved the complete verification of key middleware behavioral properties. this is a first step towards the construction of a proof-based middleware. To reach this goal, we had to improve the schizophrenic architecture in order to address both distribution functions and behavioral verification requirements.

Besides, this analysis of middleware properties provides a case study for the use of formal methods to verify a complex system. We discussed the application of advanced techniques to overcome combinatorial explosions, and how to use at best system information to optimize the verification process.

Later work will consider two main directions. The first one is to apply verification to more complex configurations (more tasks, more complex policies); but complete applications involving client and server nodes. The second direction is to increase the link between models and implementations of the μBroker. We will investigate ways to deduce code patterns from our verified formal specifications to directly produce source code for one middleware configuration. This will increase confidence in middleware as a COTS to be integrated in critical applications.

References

[1] Abrial, J., "Z: an introduction to formal methods," Cambridge University Press, 1995.

[2] Baarir, S., S. Haddad and J.-M. Ilié, *Exploiting Partial Symmetries in Well-formed nets for the Reachability and the linear Time Model Checking Problems*, in: *Proceedings of the 7th Workshop on Discrete Event Systems (WODES'04)*, Reims, France, 2004.

[3] Blair, G., L. Blair, V. Issarny, P. Tuma and A. Zarras, *The role of software architecture in constraining adaptation in component-based middleware platforms*, in: *Middleware*, 2000, pp. 164–184.

[4] Budden, T. J., *Decision Point: Will Using a COTS Component Help or Hinder Your DO-178B Certification Effort*, STSC CrossTalk, The Journal of Defense Software Engineering (2003).

[5] Chiola, G., C. Dutheillet, G. Franceschini and S. Haddad, *On Well-Formed Coloured Nets and their Symbolic Reachability Graph*, High-Level Petri Nets. Theory and Application, LNCS (1991).

[6] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, 2000.

[7] Departimento di Informatica, U. T., *Greatspn home page* (2003), http://www.di.unito.it/~greatspn/index.html.

[8] Diller, A., "The B-book," John Willey & SONS, 1994.

[9] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison Wesley, Massachusetts, 1994.

[10] Halbwachs, N., *A tutorial of Lustre* (1993).

[11] Holzmann, G., "SPIN Model Checker, The: Primer and Reference Manual," Addison Wesley Professional, 2004.

[12] Hugues, J., L. Pautet and F. Kordon, *Contributions to middleware architectures to prototype distribution infrastructures*, in: *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, San Diego, CA, USA, 2003.

[13] ISO, "Information Technology – Programming Languages – Ada," ISO, 1995, ISO/IEC/ANSI 8652:1995.

[14] Kaddour, M. and L. Pautet, *A middleware for supporting disconnections and multi-network access in mobile environments*, in: *Proceedings of the Perware workshop at the 2nd Conference on Pervasive Computing (Percom)*, Orlando, Florida, USA, 2004.

[15] Kordon, F. and E. Paviot-Adet, *Using cpn-ami to validate a safe channel protocol*, in: *Proceedings of the International Conference on Theory and Applications of Petri Nets - Tool presentation part*, Williamsburg, USA, 1999.

[16] LIP6-SRC, *Spot home page* (2003), http://spot.lip6.fr.

[17] Moody, S. A., *Distributed Objected-Oriented Real-Time Systems using a Hybrid Model of Ada 95's Built-in Distributed Capability and Emerging Real-Time CORBA Capabilities*, in: *Proceedings of the 1st IEEE International Symposium on OO RT distributed Computing*, 1998.

[18] Pyarali, I., M. Spivak, R. Cytron and D. C. Schmidt, *Evaluating and Optimizing Thread Pool Strategies for RT-CORBA*, in: *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems* (2001).

[19] Rajkumar, R., M. Gagliardi and L. Sha, *The real-time publisher/subscriber inter-process communication model for distributed real-time systems: Design and implementation*, in: *Proceeding of the First IEEE Real-Time Technology and Applications Symposium*, Denvor, Colorado, USA, 1995.

[20] Schantz, R. E. and D. C. Schmidt, *Research advances in middleware for distributed systems*, in: *Proceedings of the IFIP 17th World Computer Congress - TC6 Stream on Communication Systems: +The State of the Art* (2002), pp. 1–36.

[21] Schmidt, D. and F. Kuhns, *An overview of the real-time CORBA specification*, IEEE Computer **33** (2000), pp. 56–63.

[22] Schmidt, D., D. Levine and S. Mungee, *The design and performance of real-time object request brokers* (1998).

[23] Sharp, D. C., *Reducing avionics software cost through component based product line*, in: *Proceedings of the 10th Annual Software Technology Conference*, 1998.

[24] Soussy, Y., *Compositions of Nets via a communication medium*, in: *10th International Conference on Application and theory of Petri Nets*, Bonn, Germany, 1989.

[25] Thierry-Mieg, Y., C. Dutheillet and I. Mounier, *Automatic symmetry detection in well-formed nets*, in: *Proc. of ICATPN 2003*, Lecture Notes in Computer Science **2679** (2003), pp. 82–101.

[26] Tran, F. D. and J.-B. Stéfani, *Towards an extensible and modular ORB framework*, in: *Workshop of ECOOP'97*, Jyvaskyla, Finland, 1997.

[27] Vardi, M. Y., *An automata-theoretic approach to linear temporal logic*, in: F. Moller and G. M. Birtwistle, editors, *Proceedings of the 8th Banff Higher Order Workshop*, Lecture Notes in Computer Science **1043** (1996), pp. 238–266.

[28] Vergnaud, T., J. Hugues, L. Pautet and F. Kordon, *PolyORB: a schizophrenic middleware to build versatile reliable distributed applications*, in: *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, Palma de Mallorca, Spain, 2004.